Automatic Deployment of Software Components in the Cloud with the Aeolus Blender*

Roberto Di Cosmo
Univ Paris Diderot, Sorbonne Paris Cité, PPS, UMR 7126, CNRS and INRIA F-75205 Paris, France
roberto@dicosmo.org

Antoine Eiche
Mandriva, FR
aeiche@mandriva.com

Jacopo Mauro
Department of Computer Science and Engineering, University of Bologna, Italy
jmauro@cs.unibo.it

Gianluigi Zavattaro
Department of Computer Science and Engineering, University of Bologna, Italy
gianluigi.zavattaro@unibo.it

Stefano Zacchiroli
Univ Paris Diderot, Sorbonne Paris Cité, PPS, UMR 7126, CNRS, F-75205 Paris, France
zack@pps.univ-paris-diderot.fr

Jakub Zwolakowski
Univ Paris Diderot, Sorbonne Paris Cité, PPS, UMR 7126, CNRS, F-75205 Paris, France
jakub.zwolakowski@inria.fr

ABSTRACT

Cloud computing allows to build sophisticated software systems on virtualized infrastructures at a fraction of the cost that was necessary just a few years ago. The deployment of such complex systems, though, is still a serious issue due to the need of deploying a large number of packages and services, their elaborated interdependencies, and the need to define the (ideally optimal) allocation of software components onto available computing resources.

In this paper we present the *Aeolus Blender* (Blender in the following), a toolchain that automates the assembly and deployment of complex component-based software systems in the "cloud". By relying on a configuration optimizer and a deployment planner, Blender fully automates the deployment of real-life cloud applications on OpenStack infrastructures, by exploiting a knowledge base of software components defined in the Mandriva Armonic tool-suite. The final deployment is guaranteed to satisfy not only user requirements and software dependencies, but also to be optimal with respect to the number of used virtual machines.

1. INTRODUCTION

Automating the deployment of component-based, distributed software applications is a critical task for modern IT companies. With the advent of cloud computing, which offers the possibility to easily acquire and release computing resources, sophisticated software systems can be deployed at a fraction of the cost and time that were necessary just a few years ago. Nevertheless, the management of such applications is still a daunting task.

Several tools are used routinely to help system architects and administrators to automate at least some of the deployment and configuration phases of such complex systems. For instance, configuration managers like Puppet [24] and Chef [23] are largely used by the "DevOps" community [9] to automate the configuration of package-based applications. Domain specific languages like ConfSolve [16] or Zephyrus [7] can be used to compute—starting from a high-level partial description of the application to be realized—an (optimal) allocation of the needed software components to computing resources. Tools like Engage [11] or Metis [19] synthesizes the precise order in which low-level deployment actions should be executed to realize the desired application.

Despite the availability of such tools, the mainstream approach for deploying cloud applications is still to exploit pre-configured virtual machines images, which contain all the needed software packages and services, and that just need to be run on the target cloud system. Some examples are Bento Boxes [12], Cloud Blueprints [6], and AWS CloudFormation [1]. This approach is not entirely satisfactory though, because it does not leave room for user customization. The choices of the components to use (e.g., WordPress installed with Apache or Nginx, with NFS or GlusterFS support) lead to an explosion of configurations that can hardly be matched by the offered set of pre-configured images. Moreover, pre-configured images often force the user to run her application on specific cloud providers, inducing an undesirable vendor lock-in effect.

Arguably, the adoption of pre-configured images is still the most popular approach due to the lack of *integrated solutions* that support system designers and administrators throughout the entire process, ranging from the high-level declarative description of the application to the low-level deployment and configuration actions. In this paper we describe Blender, which is based on the approach taken in the Aeolus project [5] and strives to overcome this limitation.

*This work was supported by the French ANR project ANR-2010-SEGI-013-01 Aeolus and partially performed at IRILL, center for Free Software Research and Innovation in Paris, France, http://www.irill.org.

High-Level Architecture.

Blender realizes a software pipeline that integrates three independent tools:

Zephyrus a tool that automatically generates, starting from a partial and abstract description of the target application, a fully detailed architecture indicating which components are needed to realize such application, how to distributed them on virtual machines, and how to bind them together [7]. Zephyrus is also capable of producing optimal architectures, minimizing the amount of needed virtual machines while still guaranteeing that each software component has its needed share of computing resources (CPU power, memory, bandwidth, etc.) on the machine where it gets deployed.

Metis a planner that generates a fully detailed deployment plan that will have to be executed to bring the current state of a deployed application to the new, desired one (e.g., as produced by Zephyrus) [19]. Plans are made of individual deployment actions like installing a software component, changing its state according to its component life-cycle, provisioning virtual machines, etc. Metis relies on an *ad hoc* planning algorithm that exploits component dependencies to prune the search space and produce the needed deployment steps very efficiently (i.e., provably in polynomial time). Metis could produce plans involving hundreds of components in less than one minute.

Armonic a collection of tools that, starting from a knowledge base of information about available software components, allows for the deployment of software applications and services on several Linux distributions [20]. Each component has a list of states, and each state performs actions to deploy and configure the associated component on the target distribution.

Blender integrates the above tools, realizing a toolchain that supports system architects and administrators all the way from the design phase down to the deployment.

A declarative approach is adopted throughout Blender, according to which only a minimal amount of information needs to be initially given by the user. For instance, it is sufficient to indicate the main services the application should expose to application users, plus some deployment criteria like the level of replication to be used for critical components requiring duplication to guarantee better performances and support fault tolerance. Starting from these initial information, Blender computes the complete architecture of the application and supports the administrator in the deployment phases, during which only some configuration variables needs to be manually instantiated.

Paper structure.

In the next section we present Blender from the point of view of the users, by using it to realize a real-life, moderately complex cloud application: a replicated, load-balanced deployment of the WordPress blogging platform. Section 3 enters into more details, by showing what happens behind the scenes, at the toolchain level, when Blender is used to realize the case study of Section 2. Section 4 points to the open source implementation of Blender. Before concluding, section 5 review related literature.

Table 1: Software components used to deploy a WordPress farm

WordPress a blogging tool based on PHP;

Galera Cluster for MySQL: a multi-master cluster of MySQL databases synchronously replicated;

HAProxy a load balancer for TCP and HTTP-based applications spreading requests across multiple servers;

Varnish an HTTP accelerator designed for content-heavy dynamic web sites supporting dynamic load balancing;

HTTP Server a software component serving web server requests;

NFS client/server an application implementing a distributed file system.

2. CASE STUDY

We consider the deployment of a so-called "WordPress farm", i.e., a load balanced, replicated blogging service based on WordPress.[1] A typical approach to deploy this kind of applications on cloud infrastructures is to rely on pre-configured virtual machine images, on which someone has pre-installed various components drawn from the list shown in Figure 1, and *ad hoc* deployment scripts or recipes to realize the full architecture of the final application.

Instead, our component-based approach starts from reusable, abstract *descriptions* of these components, collected in the *Armonic* knowledge base, plus a limited amount of case-by-case information that the user will have to provide. From these elements, Blender synthesizes and then deploys a fully detailed architecture.

When executing Blender, the first piece of information the user will need to provide is an indication of the desired front-end service to be deployed, in this case the *Varnish* component.

Based on this initial piece of information, Blender will guide the user through an interactive question/answer phase, during which the types of component needed to complete the application design are chosen, and component-dependent additional information are asked to the user. The kind of information requested to the user in this second phase typically deals with desired installation policies, which usually vary on a case by case basis. For instance, as shown in Figure 1, once *Varnish* and *WordPress with NFS* is chosen, two different solutions for the database are proposed (i.e., single shared installation or multi-master replication based on *Galera*). The user can also specify that specific component pairs can not be co-installed on the same virtual machine (e.g., WordPress can not be installed with Galera for performance reasons) or that two components have to be co-installed (e.g., WordPress and HAProxy are installed on the same machine for fault tolerance reasons). This information cannot be automatically inferred, as it depends on the expected workload, so user guidance is required.

Once these pieces of information are entered, Blender translates the description of the Armonic components into the Aeolus model representations used by Zephyrus and Metis.

[1] https://wordpress.com/

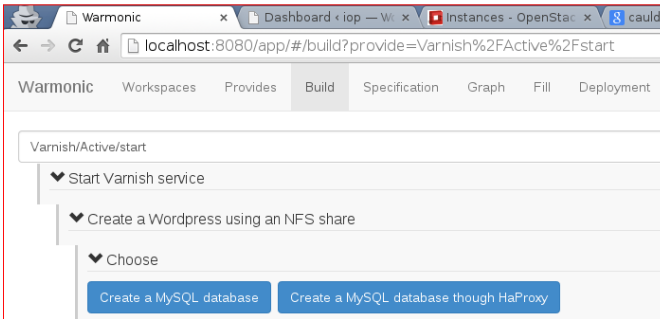

Figure 1: User inputs for WordPress installation

In particular, Zephyrus synthesizes the full architecture of the installation, indicates how many and which kind of virtual machines are needed, and distributes software components to such machines. Subsequently, Metis computes the sequence of deployment actions needed to reach the final configuration produced by Zephyrus.

The computed plan is not ready to be executed yet, because some system-level configuration parameters are still missing (e.g., administrative passwords, credentials, etc) and should be provided by the user. Blender asks the user for these information and, once all the configuration data is available, it proceeds to create the virtual machines computed by Zephyrus on the target OpenStack infrastructure. Then, Blender uses Armonic to deploy and configure components by executing state changes, according to Metis deployment plan.

In our example, during the interactive Q/A phase we have chosen Varnish to balance the traffic between 2 WordPress servers, NFS support, and 3 Galera instances. Moreover, we chose to inhibit co-installation of WordPress with Galera or the NFS Server, and to install HAProxy on every machine where WordPress is installed. The only additional piece of information asked by Blender as configuration data were the admin passwords for the DBs and HAProxy.

The final architecture produced by Blender is depicted in Figure 2. The installation requires 6 machines, 3 running Debian and 3 MBS (Mandriva Business Server).[2] It took ≈7 minutes to deploy such architecture on a simple OpenStack infrastructure deployed on an Intel Xeon server with 4 cores. The computation of the final configuration and the deployment plan was almost instantaneous: 90% of the time was spent waiting for the virtual machines to boot and for package installation.

3. AEOLUS BLENDER INTERNALS

As depicted in Figure 3, Blender is intended to be used in combination with an XMPP server and an OpenStack cloud installation. Blender is realized as an XMPP client that wraps and combines the tools Zephyrus, Metis, and Armonic and exposes its functionalities via *ad hoc* commands.[3] Basically, such commands are used to launch Zephyrus, view the graph representing the computed final configuration, fill the configuration variables, and perform the deployment actions according to the plan produced by Metis. Blender can be interacted with via a Web user interface or the command

[2] http://www.mandriva.com/en/products-services/mbs/
[3] http://xmpp.org/extensions/xep-0050.html

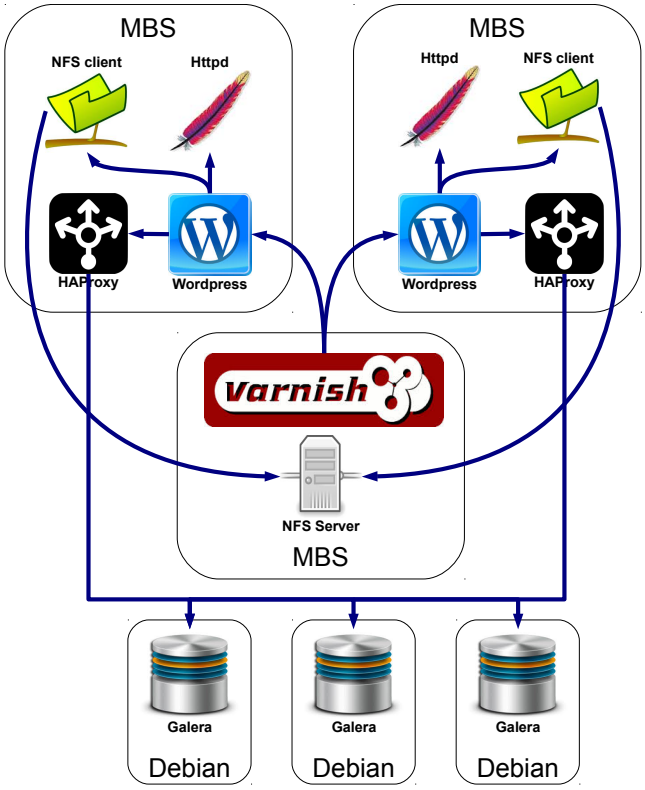

Figure 2: Computed WordPress farm architecture

line. An advantage of this architecture is that new elements can be added by wrapping them as simple XMPP clients. For instance, other IaaS offers can easily be added in addition to the currently supported OpenStack.

Blender relies on scripts that integrate Zephyrus, Metis, and Armonic following the execution flow depicted in Figure 4. Such work-flow requires two distinct inputs: an Armonic component repository, and a high-level description of the desired application to be deployed.

In Armonic a software component has an associated lifecycle that can be conceptually viewed as a state machine representing the different steps that need to be performed in order to deploy the component. For example, a component could have an associated state machine with 4 states: *not installed*, *installed*, *configured*, and *active*. Each state usually is associated to a collection of actions that need to be performed to enter into or exit each state, and actions that can be invoked on the component when a state has been entered. Technically states are implemented as Python classes, and actions are class methods. Each state has at least *enter* and *leave* hooks that are invoked when a state is entered and exited. Actions to be performed require the instantiation of a group of variables capturing information such as the required services, or the needed configuration values (with their default or optional values). In some cases, the required services should be local when the functionality must be provided in the same host where the component is deployed. For instance, in our running example, the NFS client is a local dependency of WordPress because an active WordPress

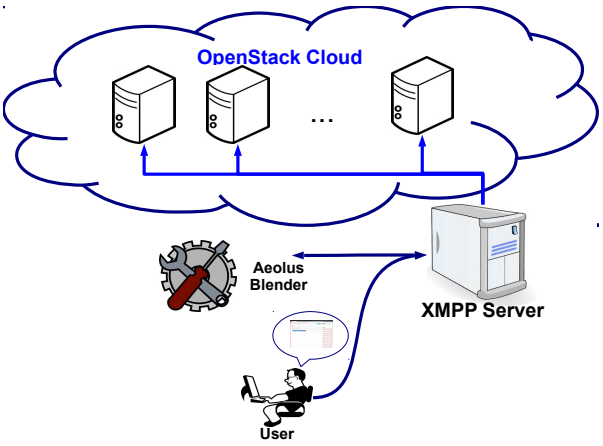

Figure 3: Blender environment

needs an NFS client to be installed on the same machine.[4]

The first step of the Blender execution flow is querying the user to gather her desiderata. This task is performed by the *Builder* that asks the user for the components she wants to install, their desired replication constraints, and information about the need or impossibility to co-install onto the same host specific pairs of components.

When the user has entered all this information, the *Builder* queries the Armonic component repository and generates:

specification file containing the encoding of the constraints that should be satisfied in the final configuration expressed in the specification language used by Zephyrus;

universe file containing the Aeolus component representations [8] of available components, in the JSON format used by both Zephyrus and Metis;

configuration data file containing indications about the system-level configured data needed to configure Armonic components. Some of them, if not already provided, will have to be entered by the user later on (e.g., credentials). Other data may be inferred from the configuration parameters of other components (e.g., WordPress can suggests a database name to its database dependency).

An excerpt of the specification file generated from user input for the running example is as follows:

```
Varnish:Active >= 1
and #(_){_ : #Galera:Active > 0 and
             #Wordpress:ActiveWithNfs > 0 } = 0
```

The first line requests a final configuration to have at least one Varnish component in the Active state. The second and third lines forbid the co-installation of Galera with WordPress. In addition to user defined constraints, Blender also requires that every component can not be installed more than once on the same virtual machine; this is currently due to an Armonic limitation that does not allow to install the same component twice on the same machine.

The *universe* file is generated by encoding Armonic components into Aeolus components, which faithfully capture

[4]for more information related to Armonic components we refer the interested reader to [21]

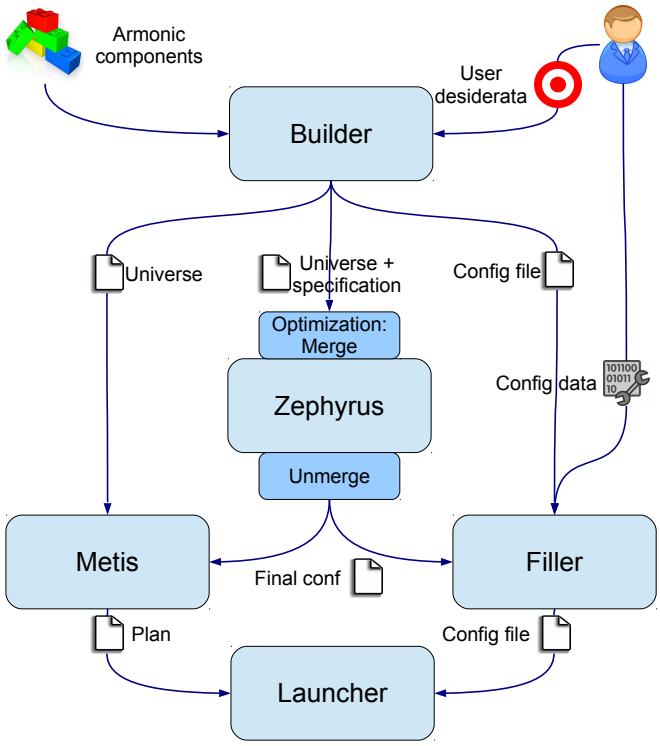

Figure 4: Blender execution flow

states and transitions. In Aeolus terminology, methods exposed by states become provide ports. These methods and special state methods (e.g., *enter* and *leave*) can expose dependencies which become require ports. As an example, a graphical representation of the Aeolus model for the WordPress component of our example is given in Figure 5. WordPress is depicted as a 5 state automaton, requiring the *add_database* functionality from the HAProxy to be configured, the *start* and *get_document_root* functionalities to be active, and the *mount* functionality from the NFS client to support the NFS. When active with NFS support, WordPress will provide the *get_website* functionality to other components.

Since the Aeolus model abstracts away from configuration data, these are stored in the *configuration data* file, which will be later used to perform deployment.

The universe file generated by the Builder is subsequently post-processed in order to merge together components that must be installed on the same machine. For instance, in our example, the WordPress components needs an NFS client to be installed on the same machine. These two components are therefore merged together obtaining a new component that consumes the sum of the resources. This simplifies the input of Zephyrus, reducing the number of components to be managed, thus speeding up the computation of the final optimal configuration, i.e., the one that uses the smallest number of virtual machines.

The solution computed by Zephyrus is then processed to decouple the components that were previously merged together. Indeed, while Zephyrus abstracts away from the internal life-cycles of the component, Metis needs to consider individual automata to compute the needed deployment actions. Metis then takes the post-processed output

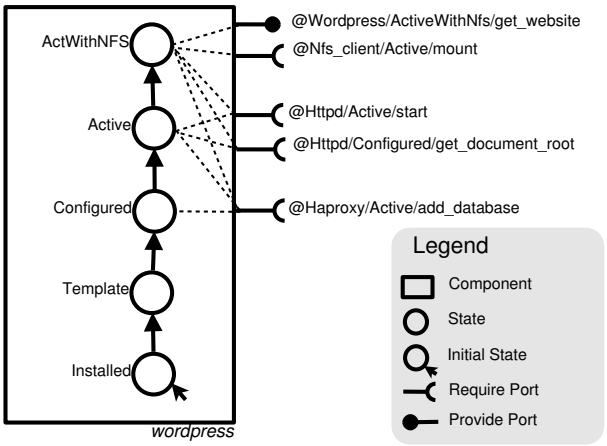

Figure 5: Aeolus representation of the WordPress component

of Zephyrus and the original Universe file to compute a deployment plan to reach the final configuration.

At this point the user is asked to provide the missing configuration data for the final deployment. The configuration data file generated by the Builder is processed together with the output of Zephyrus to detect which components should be installed and then fill the missing data querying the user if needed. This task is performed by a component dubbed *Filler* that uses several *Armonic* libraries to deduce configuration variables from default values where possible.

Once all the configuration information are filled, the plan produced by Metis and the configuration data file are passed to the *Launcher*, a Python tool that acquires and bootstraps the virtual machines indicated in the output of Zephyrus using the OpenStack API, and transforms the abstract deployment actions generated by Metis into concrete actions that are sent to Armonic agents running on individual virtual machines.

4. IMPLEMENTATION

The complete toolchain presented in this paper is publicly available and released as free software, under the GPL license. Blender consists of ≈5k lines of Python and is available from `git://git.mandriva.com/users/aeiche/aeolus-toolchain`. As Blender is an integrator, it has as software dependencies the tools it integrates:

- Zephyrus that amounts to about 10k lines of OCaml and is available from `https://github.com/aeolus-project/zephyrus/`;
- Metis that amounts to about 3.5k lines of OCaml and is available from `https://github.com/aeolus-project/metis`;
- Armonic that amounts to about 5k lines of Python, plus glue code for component life-cycles written in shell script or Augeas and is available from `https://github.com/armonic/armonic`.

Screencasts showing how Blender can be used to deploy different WordPress installations are available at `http://blog.aeolus-project.org/aeolus-blender/`.

5. RELATED WORK

The Blender toolchain integrates various tools, some of which have been discussed elsewhere. Zephyrus has been presented in [7]; the deployment experimentation done in that paper already took into account Armonic. The present paper extends [7] in several ways: it integrates Metis to drive deployment on the basis of an actual deployment plan; it adds an actual user interface turning Blender into a real, production-ready tool; and it offers tighter integration among the three tools. Thanks to Metis, which supports the synthesis of infrastructure-independent plans, Blender could also be used with other deployment engines, while deployment as described in [7] relied on hard-coded internal mechanisms of Armonic. The new user interface supports the user in lively monitor the state of the ongoing deployment, with step-by-step visualization of the effect of each deployment action. This functionality is clearly more effective if actions are executed in sequence. For this reason the current version of Blender serializes the actions synthesized by Metis (which, a priori, are maximally parallel); this limitation is arbitrary though, it could be lifted by further improving the UI.

Metis has been presented in [18]. The tool validation in that paper was done by using automatically generated descriptions of components. The integration of Metis in Blender described in this paper, on the other hand, represents the validation of Metis on real use-cases.

Different languages with their deployment engines have been proposed [10, 13, 22], but have seen limited practical adoption thus far. For this reason, as previously mentioned, the most common solution for the deployment of a cloud application is still to rely on pre-configured virtual machines [1, 6, 12]. A common, but more knowledge-intensive solution, is to use on configuration management tools, such as CFEngine [4], Puppet [24], and Chef [23]. Using those tools it is possible to declare the components that should be installed on each machine and their configuration. However, the burden of specifying where components should be deployed, and how to interconnect them is left to the system administrator.

Juju [17], supported by Canonical, includes a GUI that allows the application manager to design a final configuration and indicates the steps needed to obtain it. It cannot however compute the final and optimal configuration starting from a partial specification, nor devise an optimal order in which the deployment actions need to be performed. ConfSolve [16], like Zephyrus, relies on a constraint solver to propose an optimal allocation of virtual machines to servers, and of application components to virtual machines. However, it is unaware of system-level package incompatibilities on individual machines and does not allow to compute the actions needed to reach the computed configuration.

Similarly to Metis, Engage [11] automatically generates the right order in which deployment actions should be generated. In order to do this, Engage avoids circular component dependencies and simply perform a topological sort on the graph representing the component dependencies. Metis, on the contrary, is able to also deal with circular dependencies that can arise in practice when, for instance, configuration information flow between components in both directions (consider, e.g., a master database that first requires the slave authentication and subsequently provides the slave with a dump of the database).

Saloon [25] computes a final configuration by describing a cloud application using a feature model extended with fea-

ture cardinalities. It automatically detects inconsistencies but, differently from Zephyrus, it does not offer the ability to minimize the number of resources and virtual machines to be used.

Another relevant research direction is to leverage traditional planning techniques and tools coming from artificial intelligence. In [3,14,15] off-the-shelf planning solvers are exploited to automatically generate (re-)configuration actions. To use these tools, however, all the deployment actions with theirs preconditions and effects need to be properly specified in a formalism similar to the Planning Domain Definition Language (the *de facto* standard language for planners). The Metis approach, on the other hand, relies on simpler and natural descriptions (i.e., state machines describing the temporal order of the component configuration actions).

6. CONCLUSIONS

We have presented Blender, a tool exploiting a configurator optimizer, an *ad hoc* planner, and a deployment engine to automate installation and deployment of complex cloud applications. Blender does not rely on predefined recipes, but on reusable component descriptions that are used as building blocks to synthesize a fully functional configuration satisfying the user desiderata. Blender is easy to use, comes with a web graphical interface, and requires as input just those specific configuration parameters that cannot be deduced from the component descriptions. Blender has been validated on common deployment task, such as the installation of a *WordPress* farm. It is an open source project maintained by Mandriva that does also provide business services on top of it.

As future work we plan to test Blender on larger use cases, creating a first benchmark to be used to evaluate both the improvements of future Blender versions and for comparison with possible future competitors. In particular, new versions of Blender will reduce the deployment time by following the maximal parallelizable plan suggested by Metis. Furthermore, noticing that replicated servers (e.g., the Debian machines containing the replicated database in our *WordPress* example) share part of their deployment plan, we would like to use live virtual machine cloning instead of re-creating instances that will end up being similar from scratch.

We also plan to integrate other IaaS solutions (such as Amazon EC2, RackSpace, or Google Compute Engine) as well as other component libraries (like, e.g., those present in Juju [17] or Apache Brooklyn [2]).

7. REFERENCES

[1] Amazon. AWS CloudFormation. http://aws.amazon.com/cloudformation/.

[2] Apache Software Foundation. Apache Brooklyn. https://brooklyn.incubator.apache.org/.

[3] N. Arshad, D. Heimbigner, and A. L. Wolf. Deployment and dynamic reconfiguration planning for distributed software systems. *Software Quality Journal*, 15(3), 2007.

[4] M. Burgess. A Site Configuration Engine. *Computing Systems*, 8(2), 1995.

[5] M. Catan, R. D. Cosmo, A. Eiche, T. A. Lascu, M. Lienhardt, J. Mauro, R. Treinen, S. Zacchiroli, G. Zavattaro, and J. Zwolakowski. Aeolus: Mastering the Complexity of Cloud Application Deployment. In *ESOCC*, 2013.

[6] CenturyLink. Cloud Blueprints. http://www.centurylinkcloud.com/products/management/blueprints.

[7] R. D. Cosmo, M. Lienhardt, R. Treinen, S. Zacchiroli, J. Zwolakowski, A. Eiche, and A. Agahi. Automated synthesis and deployment of cloud applications. In *ASE*, 2014.

[8] R. D. Cosmo, J. Mauro, S. Zacchiroli, and G. Zavattaro. Aeolus: A component model for the cloud. *Inf. Comput.*, 239, 2014.

[9] DevOps. http://devops.com/.

[10] X. Etchevers, T. Coupaye, F. Boyer, and N. D. Palma. Self-Configuration of Distributed Applications in the Cloud. In *CLOUD*, 2011.

[11] J. Fischer, R. Majumdar, and S. Esmaeilsabzali. Engage: a deployment management system. In *PLDI*, 2012.

[12] Flexiant. Bento Boxes. http://www.flexiant.com/2012/12/03/application-provisioning/.

[13] G. E. Gonçalves, P. T. Endo, M. A. Santos, D. Sadok, J. Kelner, B. Melander, and J. Mångs. CloudML: An Integrated Language for Resource, Service and Request Description for D-Clouds. In *CloudCom*, 2011.

[14] H. Herry and P. Anderson. Planning with Global Constraints for Computing Infrastructure Reconfiguration. In *CP4PS*, 2012.

[15] H. Herry, P. Anderson, and G. Wickler. Automated Planning for Configuration Changes. In *LISA*, 2011.

[16] J. A. Hewson, P. Anderson, and A. D. Gordon. A Declarative Approach to Automated Configuration. In *LISA*, 2012.

[17] Juju, devops distilled. https://juju.ubuntu.com/.

[18] T. A. Lascu, J. Mauro, and G. Zavattaro. A Planning Tool Supporting the Deployment of Cloud Applications. In *ICTAI*, 2013.

[19] T. A. Lascu, J. Mauro, and G. Zavattaro. Automatic Component Deployment in the Presence of Circular Dependencies. In *FACS*, 2013.

[20] Mandriva. Armonic. http://armonic.readthedocs.org/en/latest/index.html.

[21] Mandriva. Armonic, Lifecycle anatomy. http://armonic.readthedocs.org/en/latest/lifecycle.html.

[22] OASIS. Topology and Orchestration Specification for Cloud Applications (TOSCA) Version 1.0. http://docs.oasis-open.org/tosca/TOSCA/v1.0/cs01/TOSCA-v1.0-cs01.html.

[23] Opscode. Chef. http://www.opscode.com/chef/.

[24] Puppetlabs. Puppet. http://puppetlabs.com/.

[25] C. Quinton, A. Pleuss, D. L. Berre, L. Duchien, and G. Botterweck. Consistency checking for the evolution of cardinality-based feature models. In *SPLC*, 2014.